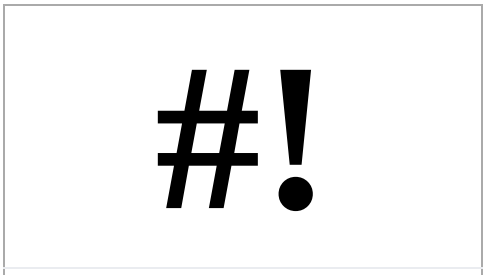
WIKIPEDIA

Shebang (Unix)



shebang

In computing, a **shebang** is the character sequence consisting of the characters number sign and exclamation mark (`#!`) at the beginning of a script. It is also called **sharp-exclamation**, **sha-bang**,[1][2] **hashbang**,[3][4] **pound-bang**,[5][6] or **hash-pling**.[7]

When a text file with a shebang is used as if it is an executable in a Unix-like operating system, the program loader mechanism parses the rest of the file's initial line as an interpreter directive. The loader executes the specified interpreter program, passing to it as an argument using the path that was initially used when attempting to run the script, so that the program may use the file as input data.[8] For example, if a script is named with the path *path/to/script*, and it starts with the following line, `#!/bin/sh`, then the program loader is instructed to run the program */bin/sh*, passing *path/to/script* as the first argument. In Linux, this behavior is the result of both kernel and user-space code.[9]

The shebang line is usually ignored by the interpreter, because the "#" character is a comment marker in many scripting languages; some language interpreters that do not use the hash mark to begin comments still may ignore the shebang line in recognition of its purpose.[10]

Contents

Syntax

The form of a shebang interpreter directive is as follows:[8]

```
#!interpreter [optional-arg]
```

in which *interpreter* is generally an absolute path to an executable program. The optional argument is a string representing a single argument. White space after `#!` is optional.

In Linux, the file specified by *interpreter* can be executed if it has one of the following:

- the execute right and contains code which the kernel can execute directly,
- a wrapper defined for it via *sysctl* (such as for executing Microsoft .exe binaries using wine),
- a shebang.

On Linux and Minix, an interpreter can also be a script. A chain of shebangs and wrappers yields a directly executable file that gets the encountered scripts as parameters in reverse order. For example, if file */bin/A* is an executable file in ELF format, file */bin/B* contains the shebang `#!/bin/A optparam`, and file */bin/C* contains the shebang `#!/bin/B`, then executing file */bin/C* resolves to `/bin/B /bin/C`, which finally resolves to `/bin/A optparam /bin/B /bin/C`.

In Solaris- and Darwin-derived operating systems (such as macOS), the file specified by *interpreter* must be an executable binary and cannot itself be a script.[11]

Examples

Some typical shebang lines:

- `#!/bin/sh` – Execute the file using the Bourne shell, or a compatible shell, assumed to be in the /bin directory
- `#!/bin/bash` – Execute the file using the Bash shell
- `#!/usr/bin/pwsh` – Execute the file using PowerShell
- `#!/usr/bin/env python3` – Execute with a Python interpreter, using the env program search path to find it
- `#!/bin/false` – Do nothing, but return a non-zero exit status, indicating failure. Used to prevent stand-alone execution of a script file intended for execution in a specific context, such as by the `.` command from sh/bash, `source` from csh/tcsh, or as a .profile, .cshrc, or .login file.

Shebang lines may include specific options that are passed to the interpreter. However, implementations vary in the parsing behavior of options; for portability, only one option should be specified without any embedded whitespace. Further portability guidelines are found below.

Purpose

Interpreter directives allow scripts and data files to be used as commands, hiding the details of their implementation from users and other programs, by removing the need to prefix scripts with their interpreter on the command line.

A Bourne shell script that is identified by the path *some/path/to/foo*, has the initial line,

```
#!/bin/sh -x
```

and is executed with parameters *bar* and *baz* as

```
some/path/to/foo bar baz
```

provides a similar result as having actually executed the following command line instead:

```
/bin/sh -x some/path/to/foo bar baz
```

If */bin/sh* specifies the Bourne shell, then the end result is that all of the shell commands in the file *some/path/to/foo* are executed with the positional variables *$1* and *$2* having the values *bar* and *baz*, respectively. Also, because the initial number sign is the character used to introduce comments in the Bourne shell language (and in the languages understood by many other interpreters), the whole shebang line is ignored by the interpreter.

However, it is up to the interpreter to ignore the shebang line; thus, a script consisting of the following two lines simply echos *both* lines to standard output when run:

```
#!/bin/cat
Hello world!
```

Strengths

When compared to the use of global association lists between file extensions and the interpreting applications, the interpreter directive method allows users to use interpreters not known at a global system level, and without administrator rights. It also allows specific selection of interpreter, without overloading the filename extension namespace (where one file extension refers to more than one file type), and allows the implementation language of a script to be changed without changing its invocation syntax by other programs. Invokers of the script need not know what the implementation language is as the script itself is responsible for specifying the interpreter to use.

Portability

Program location

Shebangs must specify absolute paths (or paths relative to current working directory) to system executables; this can cause problems on systems that have a non-standard file system layout. Even when systems have fairly standard paths, it is quite possible for variants of the same operating system to have different locations

for the desired interpreter. Python, for example, might be in */usr/bin/python3*, */usr/local/bin/python3*, or even something like */home/username/bin/python3* if installed by an ordinary user.

A similar problem exists for the POSIX shell, since POSIX only required its name to be *sh*, but did not mandate a path. A common value is `/bin/sh`, but some systems such as Solaris have the POSIX-compatible shell at */usr/xpg4/bin/sh*.[12] In many Linux systems, */bin/sh* is a hard or symbolic link to */bin/bash*, the Bourne Again shell (BASH). Using bash-specific syntax while maintaining a shebang pointing to *sh* is also not portable.[13]

Because of this it is sometimes required to edit the shebang line after copying a script from one computer to another because the path that was coded into the script may not apply on a new machine, depending on the consistency in past convention of placement of the interpreter. For this reason and because POSIX does not standardize path names, POSIX does not standardize the feature.[14] The GNU Autoconf tool can test for system support with the macro AC_SYS_INTERPRETER.[15]

Often, the program `/usr/bin/env` can be used to circumvent this limitation by introducing a level of indirection. `#!` is followed by `/usr/bin/env`, followed by the desired command without full path, as in this example:

```
#!/usr/bin/env sh
```

This mostly works because the path `/usr/bin/env` is commonly used for the `env` utility, and it invokes the first `sh` found in the user's $PATH, typically `/bin/sh`.

This still has some portability issues with OpenServer 5.0.6 and Unicos 9.0.2 which have only `/bin/env` and no `/usr/bin/env`.

Character interpretation

Another portability problem is the interpretation of the command arguments. Some systems, including Linux, do not split up the arguments;[16] for example, when running the script with the first line like,

```
#!/usr/bin/env python3 -c
```

all text after the first space is treated as a single argument, that is, `python3 -c` will be passed as one argument to `/usr/bin/env`, rather than two arguments. Cygwin also behaves this way.

Complex interpreter invocations are possible through the use of an additional wrapper. FreeBSD 6.0 (2005) introduced a `-S` option to its `env` as it changed the shebang-reading behavior to non-splitting. This option tells `env` to split the string itself.[17] The GNU `env` utility since coreutils 8.30 (2018) also includes this feature.[18] Although using this option mitigates the portability issue on the kernel end with splitting, it adds the requirement that `env` supports this particular extension.

Another problem is scripts containing a carriage return character immediately after the shebang line, perhaps as a result of being edited on a system that uses DOS line breaks, such as Microsoft Windows. Some systems interpret the carriage return character as part of the interpreter command, resulting in an error message.[19]

Magic number

The shebang is actually a human-readable instance of a magic number in the executable file, the magic byte string being `0x23 0x21`, the two-character encoding in ASCII of `#!`. This magic number is detected by the "exec" family of functions, which determine whether a file is a script or an executable binary. The presence of the shebang will result in the execution of the specified executable, usually an interpreter for the script's language. It has been claimed[20] that some old versions of Unix expect the normal shebang to be followed by a space and a slash (`#! /`), but this appears to be untrue;[21] rather, blanks after the shebang have traditionally been allowed, and sometimes documented with a space (see the 1980 email in history section below).

The shebang characters are represented by the same two bytes in extended ASCII encodings, including UTF-8, which is commonly used for scripts and other text files on current Unix-like systems. However, UTF-8 files may begin with the optional byte order mark (BOM); if the "exec" function specifically detects the bytes `0x23` and `0x21`, then the presence of the BOM (`0xEF 0xBB 0xBF`) before the shebang will prevent the script interpreter from being executed. Some authorities recommend against using the byte order mark in POSIX (Unix-like) scripts,[22] for this reason and for wider interoperability and philosophical concerns. Additionally, a byte order mark is not necessary in UTF-8, as that encoding does not have endianness issues; it serves only to identify the encoding as UTF-8.

Etymology

An executable file starting with an interpreter directive is simply called a script, often prefaced with the name or general classification of the intended interpreter. The name *shebang* for the distinctive two characters may have come from an inexact contraction of *SHArp bang* or *haSH bang*, referring to the two typical Unix names for them. Another theory on the *sh* in *shebang* is that it is from the default shell *sh*, usually invoked with shebang.[23] This usage was current by December 1989,[24] and probably earlier.

History

The shebang was introduced by Dennis Ritchie between Edition 7 and 8 at Bell Laboratories. It was also added to the BSD releases from Berkeley's Computer Science Research (present at 2.8BSD[25] and activated by default by 4.2BSD). As AT&T Bell Laboratories Edition 8 Unix, and later editions, were not released to the public, the first widely known appearance of this feature was on BSD.

The lack of an interpreter directive, but support for shell scripts, is apparent in the documentation from Version 7 Unix in 1979,[26] which describes instead a facility of the Bourne shell where files with execute permission would be handled specially by the shell, which would (sometimes depending on initial characters in the script, such as ":" or "#") spawn a subshell which would interpret and run the commands contained in the file. In this model, scripts would only behave as other commands if called from within a Bourne shell. An attempt to directly execute such a file via the operating system's own *exec()* system trap would fail, preventing scripts from behaving uniformly as normal system commands.

In later versions of Unix-like systems, this inconsistency was removed. Dennis Ritchie introduced kernel support for interpreter directives in January 1980, for Version 8 Unix, with the following description:[25]

```
From uucp Thu Jan 10 01:37:58 1980
>From dmr Thu Jan 10 04:25:49 1980 remote from research
```

```
The system has been changed so that if a file being executed
begins with the magic characters #! , the rest of the line is understood
to be the name of an interpreter for the executed file.
Previously (and in fact still) the shell did much of this job;
it automatically executed itself on a text file with executable mode
when the text file's name was typed as a command.
Putting the facility into the system gives the following
benefits.

1) It makes shell scripts more like real executable files,
because they can be the subject of 'exec.'

2) If you do a 'ps' while such a command is running, its real
name appears instead of 'sh'.
Likewise, accounting is done on the basis of the real name.

3) Shell scripts can be set-user-ID.[a]

4) It is simpler to have alternate shells available;
e.g. if you like the Berkeley csh there is no question about
which shell is to interpret a file.

5) It will allow other interpreters to fit in more smoothly.

To take advantage of this wonderful opportunity,
put

  #! /bin/sh

at the left margin of the first line of your shell scripts.
Blanks after ! are OK.  Use a complete pathname (no search is done).
At the moment the whole line is restricted to 16 characters but
this limit will be raised.
```

The feature's creator didn't give it a name, however:[28]

```
From: "Ritchie, Dennis M (Dennis)** CTR **" <dmr@[redacted]>
To: <[redacted]@talisman.org>
Date: Thu, 19 Nov 2009 18:37:37 -0600
Subject: RE: What do -you- call your #!<something> line?

 I can't recall that we ever gave it a proper name.
It was pretty late that it went in--I think that I
got the idea from someone at one of the UCB conferences
on Berkeley Unix; I may have been one of the first to
actually install it, but it was an idea that I got
from elsewhere.

As for the name: probably something descriptive like
"hash-bang" though this has a specifically British flavor, but
in any event I don't recall particularly using a pet name
for the construction.
```

Kernel support for interpreter directives spread to other versions of Unix, and one modern implementation can be seen in the Linux kernel source in *fs/binfmt_script.c*.[29]

This mechanism allows scripts to be used in virtually any context normal compiled programs can be, including as full system programs, and even as interpreters of other scripts. As a caveat, though, some early versions of kernel support limited the length of the interpreter directive to roughly 32 characters (just 16 in its first implementation), would fail to split the interpreter name from any parameters in the directive, or had other quirks. Additionally, some modern systems allow the entire mechanism to be constrained or disabled for security purposes (for example, set-user-id support has been disabled for scripts on many systems).

Note that, even in systems with full kernel support for the *#!* magic number, some scripts lacking interpreter directives (although usually still requiring execute permission) are still runnable by virtue of the legacy script handling of the Bourne shell, still present in many of its modern descendants. Scripts are then interpreted by the user's default shell.

See also

- binfmt_misc
- CrunchBang Linux
- File association
- URI fragment

Notes

a. The setuid feature is disabled in most modern operating systems following the realization that a race condition can be exploited to change the script while it's being processed.[27]

References

1. "Advanced Bash Scripting Guide: Chapter 2. Starting Off With a Sha-Bang" (http://tldp.org/LDP/abs/html/sha-bang.html). Archived (https://web.archive.org/web/20191210080709/http://tldp.org/LDP/abs/html/sha-bang.html) from the original on 10 December 2019. Retrieved 10 December 2019.
2. Cooper, Mendel (5 November 2010). *Advanced Bash Scripting Guide 5.3 Volume 1* (https://books.google.com/books?id=WPXkgFRd4OEC&q=sha-bang&pg=PA5). lulu.com. p. 5. ISBN 978-1-4357-5218-4.
3. MacDonald, Matthew (2011). *HTML5: The Missing Manual* (https://books.google.com/books?id=SR7HXy2XvBEC&pg=PA373). Sebastopol, California: O'Reilly Media. p. 373. ISBN 978-1-4493-0239-9.
4. Lutz, Mark (September 2009). *Learning Python* (https://books.google.com/books?id=1HxWGezDZcgC&pg=PA48) (4th ed.). O'Reilly Media. p. 48. ISBN 978-0-596-15806-4.
5. Guelich, Gundavaram and Birznieks, Scott, Shishir and Gunther (29 July 2000). *CGI Programming with PERL* (https://archive.org/details/cgiprogrammingwi00guel) (2nd ed.). O'Reilly Media. p. 358 (https://archive.org/details/cgiprogrammingwi00guel/page/358). ISBN 978-1-56592-419-2.
6. Lie Hetland, Magnus (4 October 2005). *Beginning Python: From Novice to Professional* (https://books.google.com/books?id=S0l1YFpRFVAC&q=pound+bang&pg=PA21). Apress. p. 21. ISBN 978-1-59059-519-0.
7. Schitka, John (24 December 2002). *Linux+ Guide to Linux Certification* (https://books.google.com/books?id=l7JhL9rJLEgC&q=hashpling&pg=PA353). Course Technology. p. 353. ISBN 978-0-619-13004-6.
8. "execve(2) - Linux man page" (http://linux.die.net/man/2/execve). Retrieved 21 October 2010.
9. Corbet, Jonathan. "The case of the supersized shebang" (https://lwn.net/Articles/779997/). *LWN.net*.

10. "SRFI 22" (http://srfi.schemers.org/srfi-22/).
11. "Python - Python3 shebang line not working as expected" (https://stackoverflow.com/questions/45444823/python3-shebang-line-not-working-as-expected).
12. "The Open Group Base Specifications Issue 7" (http://pubs.opengroup.org/onlinepubs/9699919799/utilities/sh.html#tag_20_117_16). 2008. Retrieved 5 April 2010.
13. "pixelbeat.org: Common shell script mistakes" (http://www.pixelbeat.org/programming/shell_script_mistakes.html). "It's much better to test scripts directly in a POSIX compliant shell if possible. The `bash --posix` option doesn't suffice as it still accepts some 'bashisms'"
14. "Chapter 2. Shell Command Language" (https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html), *The Open Group Base Specifications (IEEE Std 1003.1-2017)* (Issue 7 ed.), IEEE, 2018 [2008], "If the first line of a file of shell commands starts with the characters "#!", the results are unspecified."
15. *Autoconf* (https://www.gnu.org/software/autoconf/manual/autoconf-2.67/autoconf.html#System-Services), Free Software Foundation, "Macro: AC_SYS_INTERPRETER: Check whether the system supports starting scripts with a line of the form '#!/bin/sh' to select the interpreter to use for the script."
16. "/usr/bin/env behaviour" (http://mail-index.netbsd.org/netbsd-users/2008/11/09/msg002388.html). Mail-index.netbsd.org. 9 November 2008. Retrieved 18 November 2010.
17. env(1) (https://www.freebsd.org/cgi/man.cgi?query=env&sektion=1) – FreeBSD General Commands Manual
18. "env invocation" (https://www.gnu.org/software/coreutils/manual/html_node/env-invocation.html#g_t_002dS_002f_002d_002dsplit_002dstring-usage-in-scripts). *GNU Coreutils*. Retrieved 11 February 2020.
19. "Carriage Return causes bash to fail" (http://askubuntu.com/questions/372672/what-could-cause-a-script-to-fail-to-find-python-when-it-has-usr-bin-env-pyt/372691#372691). 8 November 2013.
20. "GNU Autoconf Manual v2.57, Chapter 10: Portable Shell Programming" (https://web.archive.org/web/20080118164924/http://www.gnu.org/software/autoconf/manual/autoconf-2.57/html_chapter/autoconf_10.html). Archived from the original (http://www.gnu.org/software/autoconf/manual/autoconf-2.57/html_chapter/autoconf_10.html) on 18 January 2008. Retrieved 14 May 2020.
21. "The #! magic, details about the shebang/hash-bang mechanism on various Unix flavours" (https://www.in-ulm.de/~mascheck/various/shebang/#blankrequired). Retrieved 14 May 2020.
22. "FAQ - UTF-8, UTF-16, UTF-32 & BOM: Can a UTF-8 data stream contain the BOM character (in UTF-8 form)? If yes, then can I still assume the remaining UTF-8 bytes are in big-endian order?" (http://unicode.org/faq/utf_bom.html#bom5). Retrieved 4 January 2009.
23. "Jargon File entry for shebang" (http://catb.org/jargon/html/S/shebang.html). *Catb.org*. Retrieved 16 June 2010.
24. Wall, Larry. "Perl didn't grok setuid scripts that had a space on the first line between the shebang and the interpreter name" (https://groups.google.com/group/comp.sources.bugs/msg/96bbbe2b019464c5?dmode=source&output=gplain&noredirect). USENET.
25. "CSRG Archive CD-ROMs" (http://www.mckusick.com/csrg).

26. *UNIX TIME-SHARING SYSTEM: UNIX PROGRAMMER'S MANUAL* (http://cm.bell-labs.com/7thEdMan/v7vol2a.pdf) (PDF), vol. 2A (Seventh ed.), January 1979
27. Gilles. "linux - Why is SUID disabled for shell scripts but not for binaries?" (https://security.stackexchange.com/a/194174). *Information Security Stack Exchange*.
28. Richie, Dennis. "Dennis Ritchie and Hash-Bang" (https://www.talisman.org/~erlkonig/documents/dennis-ritchie-and-hash-bang.shtml). Talisman.org. Retrieved 3 December 2020.
29. Rubini, Alessandro (31 December 1997). "Playing with Binary Formats" (http://www.linuxjournal.com/article/2568). *Linux Journal*. Retrieved 1 January 2015.

External links

- Details about the shebang mechanism on various Unix flavours (http://www.in-ulm.de/~mascheck/various/shebang/)
- #! - the Unix truth as far as I know it (http://homepages.cwi.nl/~aeb/std/hashexclam.html) (a more generic approach)
- FOLDOC shebang article (http://foldoc.org/shebang)